Dynamic Programming

- *Divide and Conquer*: Break a problem into sub-problems, solve each independently, and combine the solution.

Dynamic Programming

- *Divide and Conquer*: Break a problem into sub-problems, solve each independently, and combine the solution.
- *Dynamic Programming*: Solve a problem by breaking it into *overlapping* subproblems and solving them. Invented by Richard Bellman in the 1950s.

Dynamic Programming

- *Divide and Conquer*: Break a problem into sub-problems, solve each independently, and combine the solution.
- *Dynamic Programming*: Solve a problem by breaking it into *overlapping* subproblems and solving them. Invented by Richard Bellman in the 1950s.
- The word *Programming* here does not mean what you think.

- Dynamic Programming is applicable to problems that have:
  - Optimal substructure.
  - Overlapping subproblems.

Overlapping subproblems

- Consider algorithm to recursively calculate the n-th fibonacci number:

Input : Integer $n \geq 0$
Output: The n-th fibonacci number.

```
if n ≤ 1 then
    return 1
return fibonacciBad(n − 1) + fibonacciBad(n − 2)
```

Algorithm 1: *fibonacciBad*(n): calculates the n-th fibonacci number.

- How many calls result from *fibonacciBad*(5)?

- How many calls result from $\mathit{fibonacciBad}(5)$?
- How many are unique?

- How many calls result from *fibonacciBad*(5)?
- How many are unique?
- How can we avoid repeatedly solving *overlapping subproblems*?.

Input : Integer $n \geq 0$
Output: The n-th fibonacci number.

$M \leftarrow$ new array of size $n + 1$
for $i \leftarrow 0$ to n **do**
    $M[i] \leftarrow 0$
return $\mathit{fibRec}(n, M)$

Algorithm 2: $\mathit{fib}(n)$: Top down with *Memoization*

Input : Integer $n \geq 0$ and array $M[0 \ldots n]$
Output: The n-th fibonacci number.

if $M[n] = 0$ **then**
  if $n \leq 1$ **then**
    $M[n] = 1$
  else
    $M[n] \leftarrow \mathit{fibRec}(n-1) + \mathit{fibRec}(n-2)$
return $M[n]$

Algorithm 3: $\mathit{fibRec}(n)$: Top down with *Memoization*

Input : Integer $n \geq 0$
Output: The n-th fibonacci number.

```
M ← new array of size n + 1
M[0] ← M[1] ← 1
if n ≤ 1 then
    return M[n]
for i ← 2 to n do
    M[i] ← M[i − 1] + M[i − 2]
return M[n]
```

Algorithm 4: *fib*(n): Bottom-up.

Input : Integer $n \geq 0$
Output: The n-th fibonacci number.

$M \leftarrow$ new array of size $n + 1$
$M[0] \leftarrow M[1] \leftarrow 1$
if $n \leq 1$ **then**
    return $M[n]$
for $i \leftarrow 2$ to n **do**
    $M[i] \leftarrow M[i-1] + M[i-2]$
return $M[n]$

Algorithm 5: *fib*(n): Bottom-up.

- Time complexity?

The rod cutting problem

- Given a rod of length n, and a table showing the prices p_i for rods of sizes $1 \leq 1 \leq n$.
- What is the maximum revenue, r_n, from cutting up a rod of length n and selling the pieces.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- Strategies?

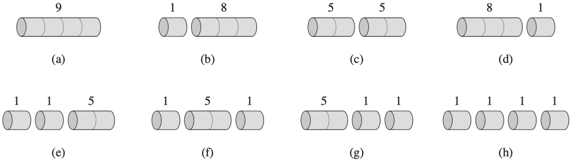
- Strategies?
- Highest price?

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

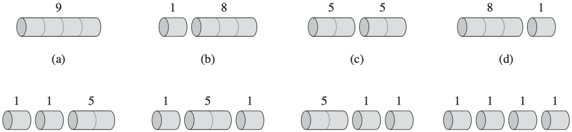
- Strategies?
- Highest price?

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- Brute force? How many ways are there to cut a rod of size 4?



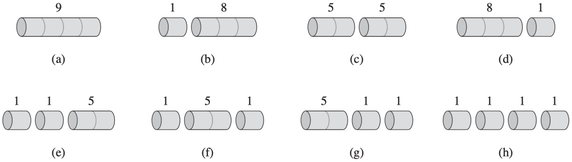
- $4 = 4$



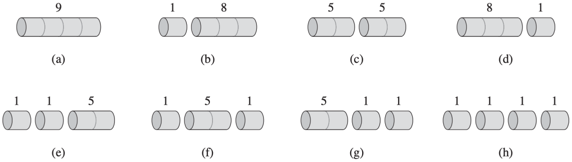
(a) (b) (c) (d)

(e) (f) (g) (h)

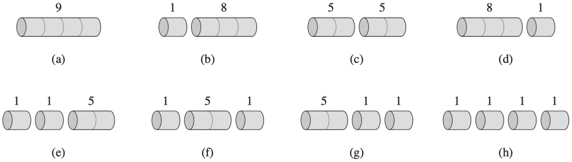
- $4 = 4$
- $4 = 1 + 3$



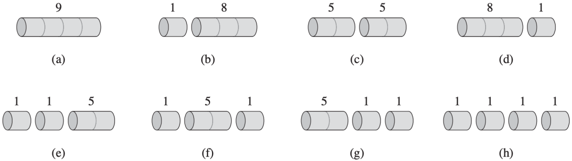
- $4 = 4$
- $4 = 1 + 3$
- $4 = 2 + 2$



- $4 = 4$
- $4 = 1 + 3$
- $4 = 2 + 2$
- $4 = 3 + 1$

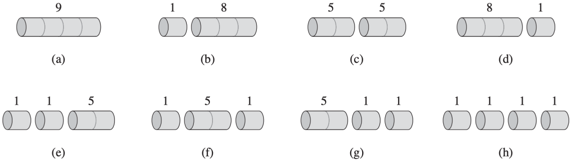


- $4 = 4$
- $4 = 1 + 3$
- $4 = 2 + 2$
- $4 = 3 + 1$
- $4 = 1 + 1 + 2$

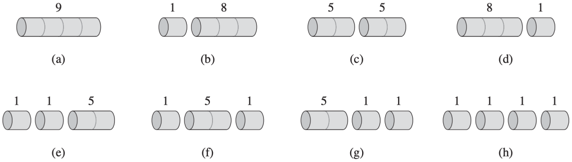


(a) (b) (c) (d) (e) (f) (g) (h)

- $4 = 4$
- $4 = 1 + 3$
- $4 = 2 + 2$
- $4 = 3 + 1$
- $4 = 1 + 1 + 2$
- $4 = 1 + 2 + 1$



- $4 = 4$
- $4 = 1 + 3$
- $4 = 2 + 2$
- $4 = 3 + 1$
- $4 = 1 + 1 + 2$
- $4 = 1 + 2 + 1$
- $4 = 2 + 1 + 1$



- $4 = 4$
- $4 = 1 + 3$
- $4 = 2 + 2$
- $4 = 3 + 1$
- $4 = 1 + 1 + 2$
- $4 = 1 + 2 + 1$
- $4 = 2 + 1 + 1$
- $4 = 1 + 1 + 1 + 1$

- Let's find r_5!

- Let's find r_5!
- Step by step.

- Let's find r_5!
- Step by step.
- Where do we cut first?

- Let's find r_5!
- Step by step.
- Where do we cut first?
  - If we cut at 1 meter, we split the rod into two pieces: 1 meter and 4 meters.

- Let's find r_5!
- Step by step.
- Where do we cut first?
  - If we cut at 1 meter, we split the rod into two pieces: 1 meter and 4 meters.
    - Maximum revenue possible $= r_1 + r_4 = p_1 + r_4 = 11$.

- Let's find r_5!
- Step by step.
- Where do we cut first?
  - If we cut at 1 meter, we split the rod into two pieces: 1 meter and 4 meters.
    - Maximum revenue possible $= r_1 + r_4 = p_1 + r_4 = 11$.
  - If we cut at 2 meters: we split the rod into 2 meters and 3 meters.

- Let's find r_5!
- Step by step.
- Where do we cut first?
  - If we cut at 1 meter, we split the rod into two pieces: 1 meter and 4 meters.
    - Maximum revenue possible $= r_1 + r_4 = p_1 + r_4 = 11$.
  - If we cut at 2 meters: we split the rod into 2 meters and 3 meters.
    - Maximum revenue possible $= r_2 + r_3$.

- Let's find r_5!
- Step by step.
- Where do we cut first?
  - If we cut at 1 meter, we split the rod into two pieces: 1 meter and 4 meters.
    - Maximum revenue possible $= r_1 + r_4 = p_1 + r_4 = 11$.
  - If we cut at 2 meters: we split the rod into 2 meters and 3 meters.
    - Maximum revenue possible $= r_2 + r_3$.
  - If we cut at 3 meters: we split the rod into 3 meters and 2 meters.

- Let's find r_5!
- Step by step.
- Where do we cut first?
  - If we cut at 1 meter, we split the rod into two pieces: 1 meter and 4 meters.
    - Maximum revenue possible $= r_1 + r_4 = p_1 + r_4 = 11$.
  - If we cut at 2 meters: we split the rod into 2 meters and 3 meters.
    - Maximum revenue possible $= r_2 + r_3$.
  - If we cut at 3 meters: we split the rod into 3 meters and 2 meters.
    - Maximum possible revenue $= r_3 + r_2$

- Let's find r_5!
- Step by step.
- Where do we cut first?
  - If we cut at 1 meter, we split the rod into two pieces: 1 meter and 4 meters.
    - Maximum revenue possible $= r_1 + r_4 = p_1 + r_4 = 11$.
  - If we cut at 2 meters: we split the rod into 2 meters and 3 meters.
    - Maximum revenue possible $= r_2 + r_3$.
  - If we cut at 3 meters: we split the rod into 3 meters and 2 meters.
    - Maximum possible revenue $= r_3 + r_2$
  - If we cut at 4 meters: we split the rod into 4 meters and 1 meters

- Let's find r_5!
- Step by step.
- Where do we cut first?
  - If we cut at 1 meter, we split the rod into two pieces: 1 meter and 4 meters.
    - Maximum revenue possible $= r_1 + r_4 = p_1 + r_4 = 11$.
  - If we cut at 2 meters: we split the rod into 2 meters and 3 meters.
    - Maximum revenue possible $= r_2 + r_3$.
  - If we cut at 3 meters: we split the rod into 3 meters and 2 meters.
    - Maximum possible revenue $= r_3 + r_2$
  - If we cut at 4 meters: we split the rod into 4 meters and 1 meters
    - Maximum revenue possible $= r_4 + r_1 = r_4 + p_1 = 11$

- Let's find r_5!
- Step by step.
- Where do we cut first?
  - If we cut at 1 meter, we split the rod into two pieces: 1 meter and 4 meters.
    - Maximum revenue possible $= r_1 + r_4 = p_1 + r_4 = 11$.
  - If we cut at 2 meters: we split the rod into 2 meters and 3 meters.
    - Maximum revenue possible $= r_2 + r_3$.
  - If we cut at 3 meters: we split the rod into 3 meters and 2 meters.
    - Maximum possible revenue $= r_3 + r_2$
  - If we cut at 4 meters: we split the rod into 4 meters and 1 meters
    - Maximum revenue possible $= r_4 + r_1 = r_4 + p_1 = 11$
  - If we don't cut the rod: Revenue $= p_5 = 10$.

- Let's find r_5!
- Step by step.
- Where do we cut first?
  - If we cut at 1 meter, we split the rod into two pieces: 1 meter and 4 meters.
    - Maximum revenue possible $= r_1 + r_4 = p_1 + r_4 = 11$.
  - If we cut at 2 meters: we split the rod into 2 meters and 3 meters.
    - Maximum revenue possible $= r_2 + r_3$.
  - If we cut at 3 meters: we split the rod into 3 meters and 2 meters.
    - Maximum possible revenue $= r_3 + r_2$
  - If we cut at 4 meters: we split the rod into 4 meters and 1 meters
    - Maximum revenue possible $= r_4 + r_1 = r_4 + p_1 = 11$
  - If we don't cut the rod: Revenue $= p_5 = 10$.
- $r_5 = max(p_5, r_1 + r_4, r_2 + r_3, r_3 + r_2, r_4 + r_1)$

- $r_n = max(p_n, r_{n-1} + r_1, r_{n-2} + r_2, \ldots, r_1 + r_{n-1})$.

- $r_n = max(p_n, r_{n-1} + r_1, r_{n-2} + r_2, \ldots, r_1 + r_{n-1})$.
- How we break the problem *matters*.

- $r_n = max(p_n, r_{n-1} + r_1, r_{n-2} + r_2, \ldots, r_1 + r_{n-1})$.
- How we break the problem *matters*.
- Problem of size n produced $2(n-1)$ subproblems.

- $r_n = max(p_n, r_{n-1} + r_1, r_{n-2} + r_2, \ldots, r_1 + r_{n-1})$.
- How we break the problem *matters*.
- Problem of size n produced $2(n-1)$ subproblems.
- What if we define it differently?

- $r_n = max(p_n, r_{n-1} + r_1, r_{n-2} + r_2, \dots, r_1 + r_{n-1})$.
- How we break the problem *matters*.
- Problem of size n produced $2(n-1)$ subproblems.
- What if we define it differently?
- Instead of deciding where the first cut will be.

- $r_n = max(p_n, r_{n-1} + r_1, r_{n-2} + r_2, \ldots, r_1 + r_{n-1})$.
- How we break the problem *matters*.
- Problem of size n produced $2(n-1)$ subproblems.
- What if we define it differently?
- Instead of deciding where the first cut will be.
- Decide where the first cut *from the left* will be.

- $r_n = max(p_n, r_{n-1} + r_1, r_{n-2} + r_2, \ldots, r_1 + r_{n-1})$.
- How we break the problem *matters*.
- Problem of size n produced $2(n-1)$ subproblems.
- What if we define it differently?
- Instead of deciding where the first cut will be.
- Decide where the first cut *from the left* will be.
- Cut, and then only cut the *right* part.

- $r_n = max(p_n, r_{n-1} + r_1, r_{n-2} + r_2, \dots, r_1 + r_{n-1})$.
- How we break the problem *matters*.
- Problem of size n produced $2(n-1)$ subproblems.
- What if we define it differently?
- Instead of deciding where the first cut will be.
- Decide where the first cut *from the left* will be.
- Cut, and then only cut the *right* part.
- $r_n = max(p_1 + r_{n-1}, p_2 + r_{n-2}, p_3 + r_{n-3}, \dots, p_n + r_0)$

- $r_n = max(p_n, r_{n-1} + r_1, r_{n-2} + r_2, \ldots, r_1 + r_{n-1})$.
- How we break the problem *matters*.
- Problem of size n produced $2(n-1)$ subproblems.
- What if we define it differently?
- Instead of deciding where the first cut will be.
- Decide where the first cut *from the left* will be.
- Cut, and then only cut the *right* part.
- $r_n = max(p_1 + r_{n-1}, p_2 + r_{n-2}, p_3 + r_{n-3}, \ldots, p_n + r_0)$
- $r_n = max_{1 \leq i \leq n}(p_i + r_{n-i})$

- $r_n = max(p_n, r_{n-1} + r_1, r_{n-2} + r_2, \ldots, r_1 + r_{n-1})$.
- How we break the problem *matters*.
- Problem of size n produced $2(n-1)$ subproblems.
- What if we define it differently?
- Instead of deciding where the first cut will be.
- Decide where the first cut *from the left* will be.
- Cut, and then only cut the *right* part.
- $r_n = max(p_1 + r_{n-1}, p_2 + r_{n-2}, p_3 + r_{n-3}, \ldots, p_n + r_0)$
- $r_n = max_{1 \leq i \leq n}(p_i + r_{n-i})$
- r_n spawns n subproblems.

- Recursive algorithm to find r_n:

Input : Size of the rod, n, and p, an array of prices
Output: The maximum revenue r_n from breaking and selling pieces

if $n = 0$ **then**
  return 0
$q \leftarrow -\infty$
for $i \leftarrow 1$ to n **do**
  $q \leftarrow max(q, p[i] + CUTROD(p, n - i))$
return q

Algorithm 6: $CUTROD(p, n)$: calculates the maximum revenue. Top-down, recursive.

- Running time:

$$T(n) = \begin{cases} 1 + \sum_{j=0}^{n-1} T(j) & n > 0 \\ 1 & n = 0 \end{cases}$$

- Is this good?

- Running time:

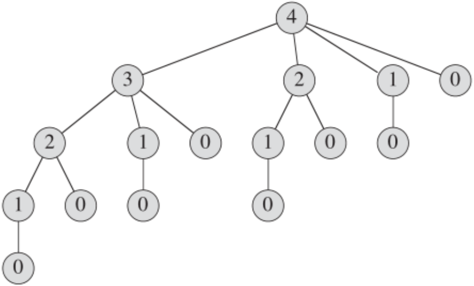
$$T(n) = \begin{cases} 1 + \sum_{j=0}^{n-1} T(j) & n > 0 \\ 1 & n = 0 \end{cases}$$

- Is this good?
- It is actually equal to 2^n.

- Running time:

$$T(n) = \begin{cases} 1 + \sum_{j=0}^{n-1} T(j) & n > 0 \\ 1 & n = 0 \end{cases}$$

- Is this good?
- It is actually equal to 2^n.
- Why is it so bad?


4
3
2
1
0
2
1
0
1
0
0
1
0
0
0
0

Dynamic Programming

- We can improve *CUTROD* by storing an array of maximum revenue values r_i for $i \leq n$.

Dynamic Programming

- We can improve *CUTROD* by storing an array of maximum revenue values r_i for $i \leq n$.
- Can either do it top-down or bottom-up.

Dynamic Programming

- We can improve *CUTROD* by storing an array of maximum revenue values r_i for $i \leq n$.
- Can either do it top-down or bottom-up.
  - Bottom-up: calculate r_1, then r_2, then $r_3, \ldots$.

Dynamic Programming

- We can improve *CUTROD* by storing an array of maximum revenue values r_i for $i \leq n$.
- Can either do it top-down or bottom-up.
  - Bottom-up: calculate r_1, then r_2, then $r_3, \ldots$.
  - Top-down: Very similar to *CUTROD*, but with *memoization*.

Memoization

Input : Size of the rod, n, and p, an array of prices
Output: The maximum revenue r_n from breaking and selling pieces

```
r[0 ... n] a new array
for i ← 0 to n do
    r[i] ← −∞
return MemoizedCutRodAux(p, n, r)
```

Algorithm 7: *MemoizedCutRod*(p, n)

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Input : Size of the rod n, p an array of prices, and r an array of revenues

Output: The maximum revenue r_n from breaking and selling pieces

if $r[n] \geq 0$ **then**
    return $r[n]$
if $n = 0$ **then**
    $q \leftarrow 0$
else
    $q \leftarrow -\infty$
    for $i \leftarrow 1$ to n **do**
        $q \leftarrow max(q, p[i] + MemoizedCutRodAux(p, n - i, r))$
$r[n] \leftarrow q$
return q

Algorithm 8: $MemoizedCutRodAux(p, n, r)$: Memoized. Recursive.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Input : Size of the rod, n, and p, an array of prices
Output: The maximum revenue r_n from breaking and selling pieces

```
let r[0 ... n] be a new array
r[0] ← 0
for j ← 1 to n  do
    q ← −∞
    for i = 1 to j do
        q ← max(q, p[i] + r[j − i])
    r[j] = q
return  r[n]
```

Algorithm 9: *BottomUpCutRod*(p, n)

- Both *MemoizedCutRod* and *BottomUpCutRod* return the

- Both *MemoizedCutRod* and *BottomUpCutRod* return the
- Maximum revenue from breaking up the rod and selling it.

- Both *MemoizedCutRod* and *BottomUpCutRod* return the
- Maximum revenue from breaking up the rod and selling it.
- But not *where* to break the rod!

- Both *MemoizedCutRod* and *BottomUpCutRod* return the
- Maximum revenue from breaking up the rod and selling it.
- But not *where* to break the rod!
- What if we want to know that?

- Both *MemoizedCutRod* and *BottomUpCutRod* return the
- Maximum revenue from breaking up the rod and selling it.
- But not *where* to break the rod!
- What if we want to know that?
- In the process of finding out the value of r_n, we can store the *decisions* that produced r_n.

- Both *MemoizedCutRod* and *BottomUpCutRod* return the
- Maximum revenue from breaking up the rod and selling it.
- But not *where* to break the rod!
- What if we want to know that?
- In the process of finding out the value of r_n, we can store the *decisions* that produced r_n.
- Let's see how to do it for *BottomUpCutRod*:

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Input : Size of the rod, n, and p, an array of prices
Output: The maximum revenue r_n from breaking and selling pieces

let $r[0 \dots n]$ and $s[0 \dots n]$ be new arrays
$r[0] \leftarrow 0$
for $j \leftarrow 1$ to n **do**
    $q \leftarrow -\infty$
    for $i = 1$ to j **do**
        if $q < p[i] + r[j-i]$ **then**
            $q \leftarrow p[i] + r[j-i]$
            $s[j] \leftarrow i$
    $r[j] = q$
return r and s

Algorithm 10: *ExtendedBottomUpCutRod*(p, n)

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Input : Size of the rod, n, and p, an array of prices
Output: How to cut up the rod to achieve the maximum revenue.

$(r, s) \leftarrow \mathit{ExtendedBottomUpCutRod}(p, n)$
while $n > 0$ **do**
    print $s[n]$
    $n \leftarrow n - s[n]$

Algorithm 11: $\mathit{PrintCutRodSolution}(p, n)$: prints the way to optimally cut the rod

Matrix Chain Multiplication

- Reminder: Two matrics A and B can be multiplied only if A has dimensions $n \times m$ and B has dimensions $m \times k$.

Matrix Chain Multiplication

- Reminder: Two matrics A and B can be multiplied only if A has dimensions $n \times m$ and B has dimensions $m \times k$.
- Problem: Given a sequence of matrices $A_1, \ldots, A_n$, we want to calculate the matrix product:

$$A_1 A_2 A_3 \cdots A_{n-1} A_n$$

- Where A_1 has dimension $p_0 \times p_1$, A_2 has dimension $p_1 \times p_2$ and so on.

Matrix Chain Multiplication

- Reminder: Two matrics A and B can be multiplied only if A has dimensions $n \times m$ and B has dimensions $m \times k$.
- Problem: Given a sequence of matrices $A_1, \ldots, A_n$, we want to calculate the matrix product:

$$A_1 A_2 A_3 \cdots A_{n-1} A_n$$

- Where A_1 has dimension $p_0 \times p_1$, A_2 has dimension $p_1 \times p_2$ and so on.
- In what order do you perform these multiplications?

Matrix Chain Multiplication

- Reminder: Two matrics A and B can be multiplied only if A has dimensions $n \times m$ and B has dimensions $m \times k$.
- Problem: Given a sequence of matrices $A_1, \ldots, A_n$, we want to calculate the matrix product:

$$A_1 A_2 A_3 \cdots A_{n-1} A_n$$

- Where A_1 has dimension $p_0 \times p_1$, A_2 has dimension $p_1 \times p_2$ and so on.
- In what order do you perform these multiplications?
- Consider the product:

$$ABC$$

- With dimensions:

Matrix Chain Multiplication

- Reminder: Two matrics A and B can be multiplied only if A has dimensions $n \times m$ and B has dimensions $m \times k$.
- Problem: Given a sequence of matrices $A_1, \ldots, A_n$, we want to calculate the matrix product:

$$A_1 A_2 A_3 \cdots A_{n-1} A_n$$

- Where A_1 has dimension $p_0 \times p_1$, A_2 has dimension $p_1 \times p_2$ and so on.
- In what order do you perform these multiplications?
- Consider the product:

$$ABC$$

- With dimensions:
  - $A : 10 \times 100$
  - $B : 100 \times 5$
  - $C : 5 \times 50$

- How many ways are there to multiply (or parenthesize) this matrix product?

- How many ways are there to multiply (or parenthesize) this matrix product?
  - $(AB)C$:
    $$10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$$
    scalar multiplications

- How many ways are there to multiply (or parenthesize) this matrix product?
  - $(AB)C$:
    $$10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$$
    scalar multiplications
  - $A(BC)$:
    $$100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$$

- How many ways are there to multiply (or parenthesize) this matrix product?
  - $(AB)C$:
    $$10 \times 100 \times 5 + 10 \times 5 \times 50 = 7500$$
    scalar multiplications
  - $A(BC)$:
    $$100 \times 5 \times 50 + 10 \times 100 \times 50 = 75000$$
- Clearly, we perform fewer operations by multiplying AB and $(AB)C$.

- For three matrices $ABCD$, we have:

- For three matrices $ABCD$, we have:
  - $((AB)C)D$.
  - $(AB)(CD)$
  - $(A(BC))D$
  - $A((BC)D)$
  - $A(B(CD))$

- So,
- Given a list of matrices $A_1, \cdots, A_n$

- So,
- Given a list of matrices $A_1, \cdots, A_n$
- How do we decide the best sequence of multiplications?
- Equivalently, how do we parenthesize the product so as to minimize the number of scalar multiplications?

- So,
- Given a list of matrices $A_1, \cdots, A_n$
- How do we decide the best sequence of multiplications?
- Equivalently, how do we parenthesize the product so as to minimize the number of scalar multiplications?
- Brute force?

- How many ways are there to multiply n matrices?

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases}$$

which is $\Omega(2^n)$, as you saw in HW 3.

- How many ways are there to multiply n matrices?

$$P(n) = \begin{cases} 1 & n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & n > 1 \end{cases}$$

which is $\Omega(2^n)$, as you saw in HW 3.

- So brute force is not feasible...

- First, we study how an optimal solution looks.

- First, we study how an optimal solution looks.
    - Consider $A_{i \dots j}$, the product of matrices A_i through A_j.
    - If $i \leq j$,

- First, we study how an optimal solution looks.
  - Consider $A_{i \ldots j}$, the product of matrices A_i through A_j.
  - If $i \leq j$,
  - There must be some k such that $A_{i \ldots k} \times A_{k+1 \ldots j}$ is performed *last*.
  - So, $A_i \cdots A_k$ is computed , and then $A_{k+1} \cdots A_j$.

- First, we study how an optimal solution looks.
  - Consider $A_{i\ldots j}$, the product of matrices A_i through A_j.
  - If $i \leq j$,
  - There must be some k such that $A_{i\ldots k} \times A_{k+1\ldots j}$ is performed *last*.
  - So, $A_i \cdots A_k$ is computed , and then $A_{k+1} \cdots A_j$.
  - Then, their product is $A_{i\ldots j}$.

- First, we study how an optimal solution looks.
  - Consider $A_{i\ldots j}$, the product of matrices A_i through A_j.
  - If $i \leq j$,
  - There must be some k such that $A_{i\ldots k} \times A_{k+1\ldots j}$ is performed *last*.
  - So, $A_i \cdots A_k$ is computed , and then $A_{k+1} \cdots A_j$.
  - Then, their product is $A_{i\ldots j}$.
  - Total cost of computing $A_{i\ldots j}$:

- First, we study how an optimal solution looks.
    - Consider $A_{i \ldots j}$, the product of matrices A_i through A_j.
    - If $i \leq j$,
    - There must be some k such that $A_{i \ldots k} \times A_{k+1 \ldots j}$ is performed *last*.
    - So, $A_i \cdots A_k$ is computed , and then $A_{k+1} \cdots A_j$.
    - Then, their product is $A_{i \ldots j}$.
    - Total cost of computing $A_{i \ldots j}$:
        - Cost of computing $A_{i \ldots k}$
        - $+$ cost of computing $A_{k+1 \ldots j}$.
        - $+$ cost of multplying $A_{i \ldots k} \times A_{k+1 \ldots j}$.

Step 1:Optimal substructure

- Suppose $A_{i\dots k} \times A_{k+1\dots j}$ is performed last in some optimal solution.

Step 1:Optimal substructure

- Suppose $A_{i \ldots k} \times A_{k+1 \ldots j}$ is performed last in some optimal solution.
- Claim: The optimal solution to $A_{i \ldots j}$ must contain optimal solutions to $A_{i \ldots k}$ and $A_{k+1 j}$.

Step 1:Optimal substructure

- Suppose $A_{i \dots k} \times A_{k+1 \dots j}$ is performed last in some optimal solution.
- Claim: The optimal solution to $A_{i \dots j}$ must contain optimal solutions to $A_{i \dots k}$ and $A_{k+1 j}$.
- Proof: the cut and paste argument!

Step 1:Optimal substructure

- Suppose $A_{i \ldots k} \times A_{k+1 \ldots j}$ is performed last in some optimal solution.
- Claim: The optimal solution to $A_{i \ldots j}$ must contain optimal solutions to $A_{i \ldots k}$ and $A_{k+1 j}$.
- Proof: the cut and paste argument!
- Suppose we find a better way to multiply $A_{i \ldots k}$

Step 1:Optimal substructure

- Suppose $A_{i \cdots k} \times A_{k+1 \cdots j}$ is performed last in some optimal solution.
- Claim: The optimal solution to $A_{i \cdots j}$ must contain optimal solutions to $A_{i \cdots k}$ and A_{k+1j}.
- Proof: the cut and paste argument!
- Suppose we find a better way to multiply $A_{i \cdots k}$
- Then, we can replace the one in our optimal solution to obtain a *better* solution.

Step 1:Optimal substructure

- Suppose $A_{i \dots k} \times A_{k+1 \dots j}$ is performed last in some optimal solution.
- Claim: The optimal solution to $A_{i \dots j}$ must contain optimal solutions to $A_{i \dots k}$ and A_{k+1j}.
- Proof: the cut and paste argument!
- Suppose we find a better way to multiply $A_{i \dots k}$
- Then, we can replace the one in our optimal solution to obtain a *better* solution.
- Contradiction.

Step 1:Optimal substructure

- Suppose $A_{i \ldots k} \times A_{k+1 \ldots j}$ is performed last in some optimal solution.
- Claim: The optimal solution to $A_{i \ldots j}$ must contain optimal solutions to $A_{i \ldots k}$ and $A_{k+1 j}$.
- Proof: the cut and paste argument!
- Suppose we find a better way to multiply $A_{i \ldots k}$
- Then, we can replace the one in our optimal solution to obtain a *better* solution.
- Contradiction.
- This *Optimal substructure* helps us construct optimal solutions to the problem by using optimal solutions to smaller subproblems.

Step 2: Recursive Solution

- How can we build an optimal solution using solutions to smaller subproblems?

Step 2: Recursive Solution

- How can we build an optimal solution using solutions to smaller subproblems?
- Our subproblems: Find the best way to parenthesize (multiply) $A_{i \ldots j}$, where $1 \leq i \leq j \leq n$.

Step 2: Recursive Solution

- How can we build an optimal solution using solutions to smaller subproblems?
- Our subproblems: Find the best way to parenthesize (multiply) $A_{i \ldots j}$, where $1 \leq i \leq j \leq n$.
- Let $m[i, j]$ be the cost of the *optimal* way to parenthesize $A_{i,j}$.

Step 2: Recursive Solution

- How can we build an optimal solution using solutions to smaller subproblems?
- Our subproblems: Find the best way to parenthesize (multiply) $A_{i \ldots j}$, where $1 \leq i \leq j \leq n$.
- Let $m[i, j]$ be the cost of the *optimal* way to parenthesize $A_{i,j}$.
- We want to find $m[i, j]$

Step 2: Recursive Solution

- How can we build an optimal solution using solutions to smaller subproblems?
- Our subproblems: Find the best way to parenthesize (multiply) $A_{i \dots j}$, where $1 \leq i \leq j \leq n$.
- Let $m[i,j]$ be the cost of the *optimal* way to parenthesize $A_{i,j}$.
- We want to find $m[i,j]$
- If we know $k \dots$

Step 2: Recursive Solution

- How can we build an optimal solution using solutions to smaller subproblems?
- Our subproblems: Find the best way to parenthesize (multiply) $A_{i\ldots j}$, where $1 \leq i \leq j \leq n$.
- Let $m[i,j]$ be the cost of the *optimal* way to parenthesize $A_{i,j}$.
- We want to find $m[i,j]$
- If we know k...
- $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$

Step 2: Recursive Solution

- How can we build an optimal solution using solutions to smaller subproblems?
- Our subproblems: Find the best way to parenthesize (multiply) $A_{i \dots j}$, where $1 \le i \le j \le n$.
- Let $m[i,j]$ be the cost of the *optimal* way to parenthesize $A_{i,j}$.
- We want to find $m[i,j]$
- If we know k...
- $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
- Since we don't know k:

$$m[i,j] = \begin{cases} 0 & i = j \\ max_{i \le k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_k p_j) & i \ne j \end{cases}$$

Step 2: Recursive Solution

- How can we build an optimal solution using solutions to smaller subproblems?
- Our subproblems: Find the best way to parenthesize (multiply) $A_{i \ldots j}$, where $1 \leq i \leq j \leq n$.
- Let $m[i,j]$ be the cost of the *optimal* way to parenthesize $A_{i,j}$.
- We want to find $m[i,j]$
- If we know k...
- $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_k p_j$
- Since we don't know k:

$$m[i,j] = \begin{cases} 0 & i = j \\ max_{i \leq k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_k p_j) & i \neq j \end{cases}$$

- We define $s[i,j] =$ the value of k for A_{ij}

Step 2: Recursive Solution

- How can we build an optimal solution using solutions to smaller subproblems?
- Our subproblems: Find the best way to parenthesize (multiply) $A_{i \dots j}$, where $1 \leq i \leq j \leq n$.
- Let $m[i,j]$ be the cost of the *optimal* way to parenthesize $A_{i,j}$.
- We want to find $m[i,j]$
- If we know k...
- $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
- Since we don't know k:

$$m[i,j] = \begin{cases} 0 & i = j \\ max_{i \leq k < j}(m[i,k] + m[k+1,j] + p_{i-1}p_kp_j) & i \neq j \end{cases}$$

- We define $s[i,j] =$ the value of k for A_{ij}
- We can now write a recursive algorithm to find $m[1,n]$ and $s[1,n]$.

Input : List of matrices $A_i \cdots A_j$, and dimensions array $p[i-1 \ldots j]$. Matrix A_r is of dimensions $p[r-1] \times p[r]$

Output: Minimum cost of multiplying the chain of matrices

1: **if** $j \leq i$ **then**
2:     **return** 0
3: $q \leftarrow \infty$
4: **for** $k \leftarrow i$ to $j-1$ **do**
5:     $q \leftarrow max(q, parenthesize1(A, i, k, p) + parenthesize1(A, k+1, j, p) + p[i-1] * p[k] * p[j]$
6: **return** q

Algorithm 12: parenthesize1(A,i,j,p)

- How bad is this?

- $$T(n) = \begin{cases} \Theta(1) & n = 1 \\ \sum_{k=1}^{n} T(i) + T(n-i) + \Theta(1) & n > 1 \end{cases}$$

- $$T(n) = \begin{cases} \Theta(1) & n = 1 \\ \sum_{k=1}^{n} T(i) + T(n-i) + \Theta(1) & n > 1 \end{cases}$$

- $\Omega(2^n)$.

-

$$T(n) = \begin{cases} \Theta(1) & n = 1 \\ \sum_{k=1}^{n} T(i) + T(n-i) + \Theta(1) & n > 1 \end{cases}$$

- $\Omega(2^n)$.
- Many subproblems are solved over and over!.

- How many subproblems are in $A_{1 \cdots n}$?

- How many subproblems are in $A_{1 \cdots n}$?
- $\binom{n}{2} + n = \frac{n(n-1)}{2} = \Theta(n^2)$ subproblems.

- How many subproblems are in $A_{1 \cdots n}$?
- $\binom{n}{2} + n = \frac{n(n-1)}{2} = \Theta(n^2)$ subproblems.
- So we can greatly improve $\Omega(2^n)$ if we don't *repeat* solving the same subproblems.

- How many subproblems are in $A_{1\dots n}$?
- $\binom{n}{2} + n = \frac{n(n-1)}{2} = \Theta(n^2)$ subproblems.
- So we can greatly improve $\Omega(2^n)$ if we don't *repeat* solving the same subproblems.
- Overlapping subproblems!

Bottom up solution

Input : List of matrices $A_1 \cdots A_n$, and dimensions array $p[0 \ldots n]$. Matrix A_r is of dimensions $p[r-1] \times p[r]$

Output: Minimum cost of multiplying the chain of matrices

let $m[1 \cdots n, 1 \cdots n]$ and $s[1 \cdots n, 1 \cdots n]$ be new arrays
for $i \leftarrow 1$ to n **do**
    $m[i,i] \leftarrow 0$
for $l \leftarrow 2$ to n **do**
    for $i \leftarrow 1$ to $n - l + 1$ **do**
        $j \leftarrow i + l - 1$
        $m[i,j] \leftarrow \infty$
        for $k \leftarrow i$ to $j - 1$ **do**
            $q \leftarrow m[i,k] + m[k+1,j] + p[i-1] \times p[k] \times p[j]$
            if $q < m[i,j]$ **then**
                $m[i,j] \leftarrow q$
                $s[i,j] \leftarrow k$
return m,s

Algorithm 13: parenthesizeBottomUp(A,n,p)

- Time complexity?

- Time complexity?
- $\Theta(n^3)$

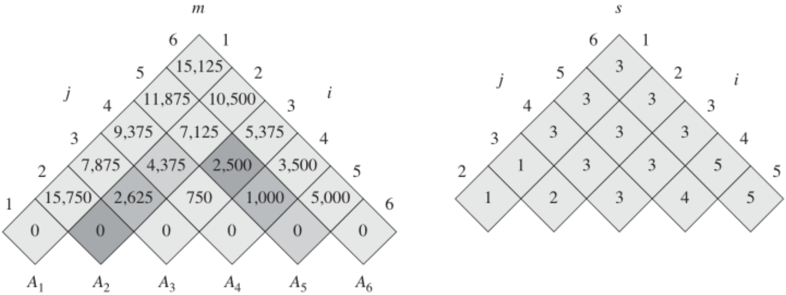


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

The tables are rotated so that the main diagonal runs horizontally. The m table uses only the main diagonal and upper triangle, and the s table uses only the upper triangle. The minimum number of scalar multiplications to multiply the 6 matrices is $m[1, 6] = 15{,}125$. Of the darker entries, the pairs that have the same shading are taken together in line 10 when computing

$$
\begin{aligned}
m[2,5] &= \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 &= 0 + 2500 + 35 \cdot 15 \cdot 20 &= 13{,}000 , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 &= 2625 + 1000 + 35 \cdot 5 \cdot 20 &= 7125 , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 &= 4375 + 0 + 35 \cdot 10 \cdot 20 &= 11{,}375 \end{cases} \\
&= 7125 .
\end{aligned}
$$

```
MEMOIZED-MATRIX-CHAIN(p)
n = p.length - 1
let m[1..n, 1..n] be a new table
for i = 1 to n
    for j = i to n
        m[i, j] = ∞
return LOOKUP-CHAIN(m, p, 1, n)
```

```
LOOKUP-CHAIN(m, p, i, j)
if m[i, j] < ∞
    return m[i, j]
if i == j
    m[i, j] = 0
else for k = i to j - 1
        q = LOOKUP-CHAIN(m, p, i, k)
            + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i-1} p_k p_j
        if q < m[i, j]
            m[i, j] = q
return m[i, j]
```

- Why is it so much better than the $\Omega(2^n)$ algorithm?

- Why is it so much better than the $\Omega(2^n)$ algorithm?
- How can we analyze the memoized, top-down version?

- Why is it so much better than the $\Omega(2^n)$ algorithm?
- How can we analyze the memoized, top-down version?
- We can, if we understand how line 6 of LOOKUP-CHAIN behaves.

- Why is it so much better than the $\Omega(2^n)$ algorithm?
- How can we analyze the memoized, top-down version?
- We can, if we understand how line 6 of LOOKUP-CHAIN behaves.
- This line calls the method twice. There are two types of calls that can happen here.

- Why is it so much better than the $\Omega(2^n)$ algorithm?
- How can we analyze the memoized, top-down version?
- We can, if we understand how line 6 of LOOKUP-CHAIN behaves.
- This line calls the method twice. There are two types of calls that can happen here.
  1. Calls to solve A_{ij} the first time.

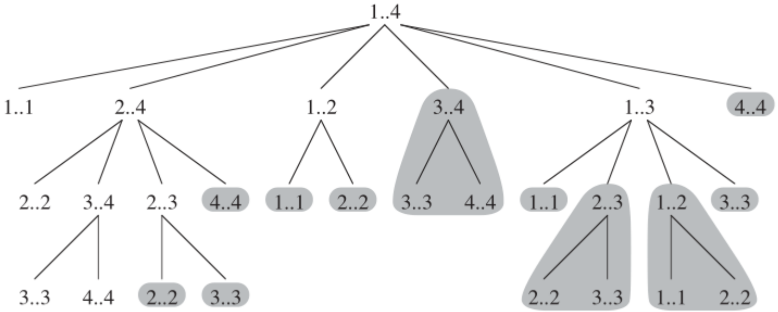
- Why is it so much better than the $\Omega(2^n)$ algorithm?
- How can we analyze the memoized, top-down version?
- We can, if we understand how line 6 of LOOKUP-CHAIN behaves.
- This line calls the method twice. There are two types of calls that can happen here.
    1. Calls to solve A_{ij} the first time.
    2. Calls to solve already solve problems.

- Why is it so much better than the $\Omega(2^n)$ algorithm?
- How can we analyze the memoized, top-down version?
- We can, if we understand how line 6 of LOOKUP-CHAIN behaves.
- This line calls the method twice. There are two types of calls that can happen here.
  1. Calls to solve A_{ij} the first time.
  2. Calls to solve already solve problems.
- Of the first type, there are $\Theta(n^2)$.

- Why is it so much better than the $\Omega(2^n)$ algorithm?
- How can we analyze the memoized, top-down version?
- We can, if we understand how line 6 of LOOKUP-CHAIN behaves.
- This line calls the method twice. There are two types of calls that can happen here.
    1. Calls to solve A_{ij} the first time.
    2. Calls to solve already solve problems.
- Of the first type, there are $\Theta(n^2)$.
- Each call of the first type calls $LOOKUP - CHAIN$ this many times: $j - i = O(n)$.

- Why is it so much better than the $\Omega(2^n)$ algorithm?
- How can we analyze the memoized, top-down version?
- We can, if we understand how line 6 of LOOKUP-CHAIN behaves.
- This line calls the method twice. There are two types of calls that can happen here.
  1. Calls to solve A_{ij} the first time.
  2. Calls to solve already solve problems.
- Of the first type, there are $\Theta(n^2)$.
- Each call of the first type calls $LOOKUP - CHAIN$ this many times: $j - i = O(n)$.
- So, there is $O(n^3)$ calls of the second type.

- Why is it so much better than the $\Omega(2^n)$ algorithm?
- How can we analyze the memoized, top-down version?
- We can, if we understand how line 6 of LOOKUP-CHAIN behaves.
- This line calls the method twice. There are two types of calls that can happen here.
    1. Calls to solve A_{ij} the first time.
    2. Calls to solve already solve problems.
- Of the first type, there are $\Theta(n^2)$.
- Each call of the first type calls $LOOKUP - CHAIN$ this many times: $j - i = O(n)$.
- So, there is $O(n^3)$ calls of the second type.
- We can conclude the algorithm is $O(n^3)$.


1..4
1..1
2..4
1..2
3..4
1..3
4..4
2..2
3..4
2..3
4..4
1..1
2..2
3..3
4..4
1..1
2..3
1..2
3..3
3..3
4..4
2..2
3..3
2..2
3..3
1..1
2..2

- How do we construct a parenthesization from s?

- How do we construct a parenthesization from s?
- $s[i, j]$ stores which k to pick for $A_{i \dots j}$.

- How do we construct a parenthesization from s?
- $s[i, j]$ stores which k to pick for $A_{i \dots j}$.
- $s[1, n]$ stores which k to pick for $A_{1 \dots n}$.

- How do we construct a parenthesization from s?
- $s[i,j]$ stores which k to pick for $A_{i \cdots j}$.
- $s[1,n]$ stores which k to pick for $A_{1 \cdots n}$.
- Which means we break A_{1n} into $A_{1 \cdots s[1,n]} A_{s[1,n]} A_{s[1,n]+1 \cdots n}$

- How do we construct a parenthesization from s?
- $s[i, j]$ stores which k to pick for $A_{i \cdots j}$.
- $s[1, n]$ stores which k to pick for $A_{1 \cdots n}$.
- Which means we break A_{1n} into $A_{1 \cdots s[1,n]} A_{s[1,n]} A_{s[1,n]+1 \cdots n}$
- The multiplications previous to that? at $s[1, s[1, n]]$ and $s[s[1, n] + 1, n]$.

- How do we construct a parenthesization from s?
- $s[i, j]$ stores which k to pick for $A_{i \cdots j}$.
- $s[1, n]$ stores which k to pick for $A_{1 \cdots n}$.
- Which means we break A_{1n} into $A_{1 \cdots s[1,n]} A_{s[1,n]} A_{s[1,n]+1 \cdots n}$
- The multiplications previous to that? at $s[1, s[1, n]]$ and $s[s[1, n] + 1, n]$.
- Recursively:

Input : Table s and indices i and j.
Output: Prints parenthesization.

if $i = j$ **then**
    print A_i
else
    print (
    printParens(s, i, s[i,j])
    printParens(s, s[i,j]+1, j)
    print)

Algorithm 14: printParens(s, i, j)

Revision: DP basics

A problem may have a DP solution if it has:

- Optimal substructure. Identified usually by the steps:

Revision: DP basics

A problem may have a DP solution if it has:

- Optimal substructure. Identified usually by the steps:
    1. Solving the problem involves some *choice*, which breaks it into smaller subproblems.

Revision: DP basics

A problem may have a DP solution if it has:

- Optimal substructure. Identified usually by the steps:
  1. Solving the problem involves some *choice*, which breaks it into smaller subproblems.
  2. Suppose that this choice is *given*,

Revision: DP basics

A problem may have a DP solution if it has:

- Optimal substructure. Identified usually by the steps:
    1. Solving the problem involves some *choice*, which breaks it into smaller subproblems.
    2. Suppose that this choice is *given*,
    3. Describe the subproblems resulting,

Revision: DP basics

A problem may have a DP solution if it has:

- Optimal substructure. Identified usually by the steps:
  1. Solving the problem involves some *choice*, which breaks it into smaller subproblems.
  2. Suppose that this choice is *given*,
  3. Describe the subproblems resulting,
  4. Show that the optimal solution must contain optimal solutions to the subproblems. Use the *cut and paste* argument.

Revision: DP basics

A problem may have a DP solution if it has:

- Optimal substructure. Identified usually by the steps:
  1. Solving the problem involves some *choice*, which breaks it into smaller subproblems.
  2. Suppose that this choice is *given*,
  3. Describe the subproblems resulting,
  4. Show that the optimal solution must contain optimal solutions to the subproblems. Use the *cut and paste* argument.
- Overlapping subproblems. If you avoid repeatedly solving subproblems, your running time will be determined by

Revision: DP basics

A problem may have a DP solution if it has:

- Optimal substructure. Identified usually by the steps:
  1. Solving the problem involves some *choice*, which breaks it into smaller subproblems.
  2. Suppose that this choice is *given*,
  3. Describe the subproblems resulting,
  4. Show that the optimal solution must contain optimal solutions to the subproblems. Use the *cut and paste* argument.
- Overlapping subproblems. If you avoid repeatedly solving subproblems, your running time will be determined by
  - The number of unique subproblems

Revision: DP basics

A problem may have a DP solution if it has:

- Optimal substructure. Identified usually by the steps:
  1. Solving the problem involves some *choice*, which breaks it into smaller subproblems.
  2. Suppose that this choice is *given*,
  3. Describe the subproblems resulting,
  4. Show that the optimal solution must contain optimal solutions to the subproblems. Use the *cut and paste* argument.
- Overlapping subproblems. If you avoid repeatedly solving subproblems, your running time will be determined by
  - The number of unique subproblems
  - The cost of solving one subproblem.

Building a DP algorithm

1. Describe the problem's solution in a general way that can also describe the solutions to subproblems.

Building a DP algorithm

1. Describe the problem's solution in a general way that can also describe the solutions to subproblems.
2. Find the optimal substructure and define it concretely.

Building a DP algorithm

1. Describe the problem's solution in a general way that can also describe the solutions to subproblems.
2. Find the optimal substructure and define it concretely.
3. Use the optimal substructure to recursively define the optimal solution to the problem in terms of subproblems.

Building a DP algorithm

1. Describe the problem's solution in a general way that can also describe the solutions to subproblems.
2. Find the optimal substructure and define it concretely.
3. Use the optimal substructure to recursively define the optimal solution to the problem in terms of subproblems.
4. Identify the dimensions and values for a table with one entry for each subproblem.

Building a DP algorithm

1. Describe the problem's solution in a general way that can also describe the solutions to subproblems.
2. Find the optimal substructure and define it concretely.
3. Use the optimal substructure to recursively define the optimal solution to the problem in terms of subproblems.
4. Identify the dimensions and values for a table with one entry for each subproblem.
5. Give either a bottom up or top-down memoization algorithm.

Building a DP algorithm

1. Describe the problem's solution in a general way that can also describe the solutions to subproblems.
2. Find the optimal substructure and define it concretely.
3. Use the optimal substructure to recursively define the optimal solution to the problem in terms of subproblems.
4. Identify the dimensions and values for a table with one entry for each subproblem.
5. Give either a bottom up or top-down memoization algorithm.
6. Now your algorithm can find the *value* of the optimal solution.

Building a DP algorithm

1. Describe the problem's solution in a general way that can also describe the solutions to subproblems.
2. Find the optimal substructure and define it concretely.
3. Use the optimal substructure to recursively define the optimal solution to the problem in terms of subproblems.
4. Identify the dimensions and values for a table with one entry for each subproblem.
5. Give either a bottom up or top-down memoization algorithm.
6. Now your algorithm can find the *value* of the optimal solution.
7. Modify your algorithm to also keep track of choices leading to that optimal solution value, in a second table.

Building a DP algorithm

1. Describe the problem's solution in a general way that can also describe the solutions to subproblems.
2. Find the optimal substructure and define it concretely.
3. Use the optimal substructure to recursively define the optimal solution to the problem in terms of subproblems.
4. Identify the dimensions and values for a table with one entry for each subproblem.
5. Give either a bottom up or top-down memoization algorithm.
6. Now your algorithm can find the *value* of the optimal solution.
7. Modify your algorithm to also keep track of choices leading to that optimal solution value, in a second table.
8. Using the second table, construct an optimal solution.

The Knapsack Problem

- A thief breaks into a vault. There are n number of items.
- Each itme has a value v_i, and a weight w_i.

The Knapsack Problem

- A thief breaks into a vault. There are n number of items.
- Each itme has a value v_i, and a weight w_i.
- The thief wants to take as much total value as possible.

The Knapsack Problem

- A thief breaks into a vault. There are n number of items.
- Each itme has a value v_i, and a weight w_i.
- The thief wants to take as much total value as possible.
- However, the thief only brought a knapsack of limited total weight capacity, W.

The Knapsack Problem

- A thief breaks into a vault. There are n number of items.
- Each itme has a value v_i, and a weight w_i.
- The thief wants to take as much total value as possible.
- However, the thief only brought a knapsack of limited total weight capacity, W.
- The solution to this instance of the knapsack problem:
- The subset of items, S, the thief can take such that:

The Knapsack Problem

- A thief breaks into a vault. There are n number of items.
- Each itme has a value v_i, and a weight w_i.
- The thief wants to take as much total value as possible.
- However, the thief only brought a knapsack of limited total weight capacity, W.
- The solution to this instance of the knapsack problem:
- The subset of items, S, the thief can take such that:
  - The total stolen value $\sum_{i \in S} v_i$ is maximized.
  - The total weight of stolen goods $sum_{i \in S} w_i$ does not exceed the knapsack's capacity, W.

- Consider the example:

	Item 1	Item 2	Item 3
v_i	4	4	6
w_i	1	2	4

- Does this problem have optimal substructure?
- How can we see it?

- Does this problem have optimal substructure?
- How can we see it?
- Consider an optimal solution in the form of the set S.

- Does this problem have optimal substructure?
- How can we see it?
- Consider an optimal solution in the form of the set S.
- Consider the *value* of S:

$$m(S, W) = \sum_{i \in S} v_i$$

- Does this problem have optimal substructure?
- How can we see it?
- Consider an optimal solution in the form of the set S.
- Consider the *value* of S:

$$m(S, W) = \sum_{i \in S} v_i$$

- This value is optimal (maximum), and respecting the total capacity W.

- Does this problem have optimal substructure?
- How can we see it?
- Consider an optimal solution in the form of the set S.
- Consider the *value* of S:

$$m(S, W) = \sum_{i \in S} v_i$$

- This value is optimal (maximum), and respecting the total capacity W.
- If we remove one element, v_k from S,

- Does this problem have optimal substructure?
- How can we see it?
- Consider an optimal solution in the form of the set S.
- Consider the *value* of S:

$$m(S, W) = \sum_{i \in S} v_i$$

- This value is optimal (maximum), and respecting the total capacity W.
- If we remove one element, v_k from S,
- We get the set $S \setminus \{v_k\}$, an optimal solution for which knapsack problem instance?

- Does this problem have optimal substructure?
- How can we see it?
- Consider an optimal solution in the form of the set S.
- Consider the *value* of S:

$$m(S, W) = \sum_{i \in S} v_i$$

- This value is optimal (maximum), and respecting the total capacity W.
- If we remove one element, v_k from S,
- We get the set $S \setminus \{v_k\}$, an optimal solution for which knapsack problem instance?
- $m(S \setminus \{v_k\}, W - w_k)$.

- Does this problem have optimal substructure?
- How can we see it?
- Consider an optimal solution in the form of the set S.
- Consider the *value* of S:

$$m(S, W) = \sum_{i \in S} v_i$$

- This value is optimal (maximum), and respecting the total capacity W.
- If we remove one element, v_k from S,
- We get the set $S \setminus \{v_k\}$, an optimal solution for which knapsack problem instance?
- $m(S \setminus \{v_k\}, W - w_k)$.
- In other words, $m(S, W) = m(S \setminus \{v_k\}, W - w_k) + v_k$

- Recursive definition of $m(S, W)$?

- Recursive definition of $m(S, W)$?
-

$$m(S, W) = \begin{cases} 0 & S = \emptyset \\ \max_{i \in S : w_i \leq W} m(v_i + S \setminus \{i\}, W - w_i) & \text{otherwise} \end{cases}$$

- Recursive definition of $m(S, W)$?
- $$m(S, W) = \begin{cases} 0 & S = \emptyset \\ \max_{i \in S : w_i \leq W} m(v_i + S \setminus \{i\}, W - w_i) & \text{otherwise} \end{cases}$$
- How many rows would we need to represent $m(S, W)$??

- Recursive definition of $m(S, W)$?
-

$$m(S, W) = \begin{cases} 0 & S = \emptyset \\ \max_{i \in S : w_i \leq W} m(v_i + S \setminus \{i\}, W - w_i) & \text{otherwise} \end{cases}$$

- How many rows would we need to represent $m(S, W)$??
- One row for every subset. Exponential in n.

- Recursive definition of $m(S, W)$?
-

$$m(S, W) = \begin{cases} 0 & S = \emptyset \\ \max_{i \in S : w_i \leq W} m(v_i + S \setminus \{i\}, W - w_i) & \text{otherwise} \end{cases}$$

- How many rows would we need to represent $m(S, W)$??
- One row for every subset. Exponential in n.
- Our optimal substructure is correct, but...

- Recursive definition of $m(S, W)$?
-

$$m(S, W) = \begin{cases} 0 & S = \emptyset \\ \max_{i \in S : w_i \leq W} m(v_i + S \setminus \{i\}, W - w_i) & \text{otherwise} \end{cases}$$

- How many rows would we need to represent $m(S, W)$??
- One row for every subset. Exponential in n.
- Our optimal substructure is correct, but...
- We need to take a step back and think of a better way to represent the subproblems and their solution.

- Instead of indexing each subproblem with S and W, we can vastly decrease the number of subproblems by

- Instead of indexing each subproblem with S and W, we can vastly decrease the number of subproblems by
- Indexing each problem by integers i and j, as follows:

- Instead of indexing each subproblem with S and W, we can vastly decrease the number of subproblems by
- Indexing each problem by integers i and j, as follows:
- Think of the set of all items as $\{1, 2, \ldots, n\}$

- Instead of indexing each subproblem with S and W, we can vastly decrease the number of subproblems by
- Indexing each problem by integers i and j, as follows:
- Think of the set of all items as $\{1, 2, \ldots, n\}$
- Then, we will define our original problem as finding the solution whose value is $m(n, W)$, where W is the capacity.

- Instead of indexing each subproblem with S and W, we can vastly decrease the number of subproblems by
- Indexing each problem by integers i and j, as follows:
- Think of the set of all items as $\{1, 2, \ldots, n\}$
- Then, we will define our original problem as finding the solution whose value is $m(n, W)$, where W is the capacity.
- $m(n, W)$: *value* of the optimal solution for items $\{1, 2, \ldots, n\}$ and knapsack capacity W.

- For $m(n, W)$, either:

- For $m(n, W)$, either:
  1. The item n is in the optimal subset , and $m(n, W) = v_n + m(n-1, W - w_n)$,

- For $m(n, W)$, either:
  1. The item n is in the optimal subset , and $m(n, W) = v_n + m(n-1, W - w_n)$,
  2. The item n is not in the optimal subset, and $m(n, W) = m(n-1, W)$.

- For $m(n, W)$, either:
  1. The item n is in the optimal subset , and $m(n, W) = v_n + m(n-1, W - w_n)$,
  2. The item n is not in the optimal subset, and $m(n, W) = m(n-1, W)$.
- We can use this to define $m(i, j)$ recursively:

$$m(i,j) = \begin{cases} 0 & i = 0 \\ m(i-1,j) & w_i > j \\ max\{v_i + m(i-1, j-w_i), m(i-1,j)\} & w_i \leq j \end{cases}$$

- For $m(n, W)$, either:
  1. The item n is in the optimal subset , and $m(n, W) = v_n + m(n-1, W - w_n)$,
  2. The item n is not in the optimal subset, and $m(n, W) = m(n-1, W)$.
- We can use this to define $m(i, j)$ recursively:

$$m(i,j) = \begin{cases} 0 & i = 0 \\ m(i-1,j) & w_i > j \\ max\{v_i + m(i-1, j-w_i), m(i-1,j)\} & w_i \leq j \end{cases}$$

- Bottom up algorithm to find $m(n, W)$?

- For $m(n, W)$, either:
    1. The item n is in the optimal subset , and $m(n, W) = v_n + m(n-1, W-w_n)$,
    2. The item n is not in the optimal subset, and $m(n, W) = m(n-1, W)$.
- We can use this to define $m(i, j)$ recursively:

$$m(i,j) = \begin{cases} 0 & i = 0 \\ m(i-1,j) & w_i > j \\ max\{v_i + m(i-1, j-w_i), m(i-1,j)\} & w_i \leq j \end{cases}$$

- Bottom up algorithm to find $m(n, W)$?
- How many rows and columns in m?

Input : List of values $\{v_1, v_2, \ldots, v_n\}$,integer weights $\{w_1, w_2, \ldots, w_n\}$, and integer capacity W

Output: Value of optimal solution to the 0,1-knapsack problem.

```
for i ← 1 to n do
    m[i,0] ← 0
for j ← 1 to W do
    m[0,j] ← 0
for i ← 1 to n do
    for j ← 1 to W do
        if w_j > j then
            m[i,j] ← m[i−1,j]
        else
            m[i,j] ← max{v[i] + m[i−1, j−w[i]], m[i−1,j]}
return m
```

- Consider the exmaple, with capacity $W = 5$:

item	weight	value
1	2	12
2	1	10
3	3	20
4	2	15

		0	1	2	3	4	5
	0						
$w_1 = 2, v_1 = 12$	1						
$w_2 = 1, v_2 = 10$	2						
$w_3 = 3, v_3 = 20$	3						
$w_4 = 2, v_4 = 15$	4						

		0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0					
$w_2 = 1, v_2 = 10$	2	0					
$w_3 = 3, v_3 = 20$	3	0					
$w_4 = 2, v_4 = 15$	4	0					

		0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0					
$w_3 = 3, v_3 = 20$	3	0					
$w_4 = 2, v_4 = 15$	4	0					

		0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0					
$w_4 = 2, v_4 = 15$	4	0					

		0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0					

		0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	37

- Time complexity?

- Time complexity?
- $\Theta(nW)$: Is this linear? Is this good?

- Time complexity?
- $\Theta(nW)$: Is this linear? Is this good?
- Linear in W, not in the size of input n.

- Time complexity?
- $\Theta(nW)$: Is this linear? Is this good?
- Linear in W, not in the size of input n.
- This can be worse than exponential!

- Time complexity?
- $\Theta(nW)$: Is this linear? Is this good?
- Linear in W, not in the size of input n.
- This can be worse than exponential!
- Can we improve that? Not by much, but we can.

- Time complexity?
- $\Theta(nW)$: Is this linear? Is this good?
- Linear in W, not in the size of input n.
- This can be worse than exponential!
- Can we improve that? Not by much, but we can.
- Top-down approach with memoization does not solve *all* the subproblems.

if $m[i, j] < 0$ **then**
    if $w[i] > j$ **then**
        $m[i, j] \leftarrow m[i - 1, j]$
    else
        $m[i, j] \leftarrow max\{v[i] + m[i - 1, j - w[i]], m[i - 1, j]\}$
return $m[i, j]$

- How can we construct a solution from m?

$i \leftarrow n$
$j \leftarrow W$
while $i > 0$ **do**
    if $w[i] \leq j$ and $v[i] + m[i-1, j - w[i]] > m[i-1, j]$ **then**
        Print i
        $i \leftarrow i - 1$
        $j \leftarrow j - w[j]$
    else
        $i \leftarrow i - 1$